モジュール機構の最前線
ー OOP, AOP, さてその次は？

千葉　滋
東京工業大学

PPL Summer School 2009

モジュール化

- 目的
  - コードの再利用
    - フレームワーク化
  - 保守性の向上
    - チームによる分担開発
    - 仕様と実装の分離

Abstract Data Type

- CLU [Liskov 1977]
  - 型に別名をつける
  - その型の値に対して適用可能な関数を定義
    - それらの関数以外は適用不可

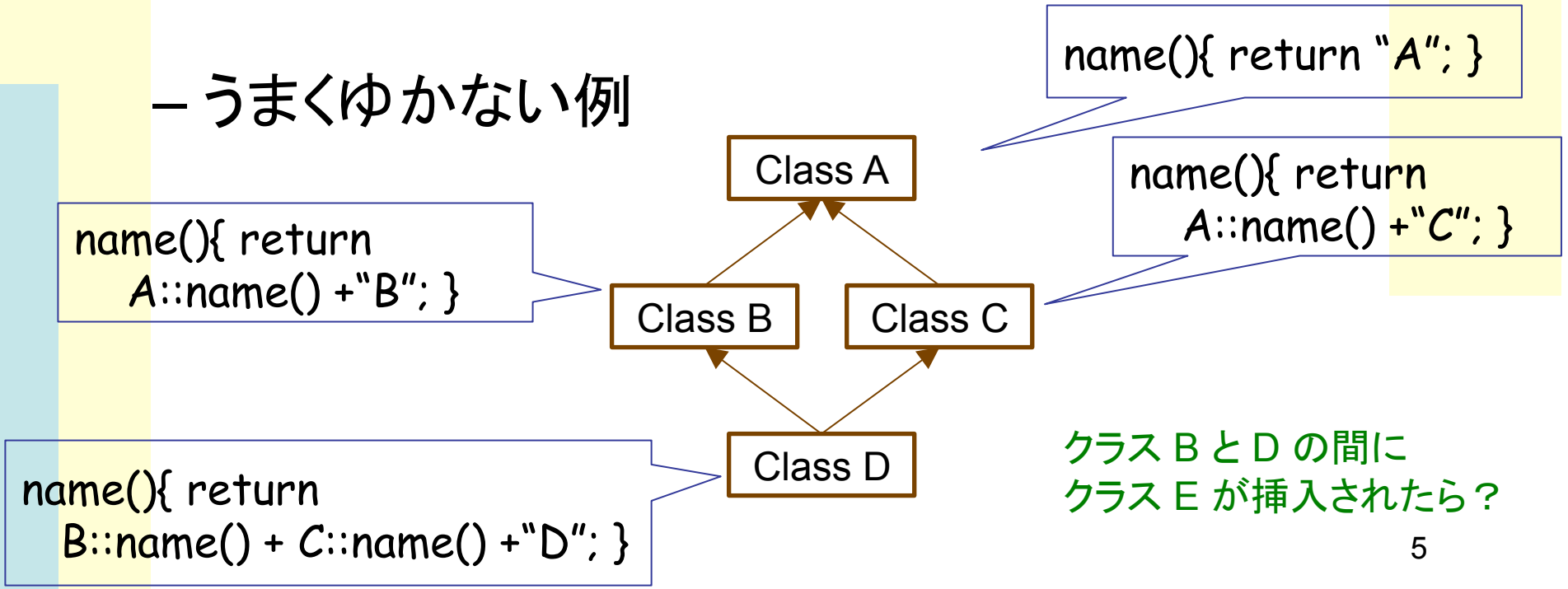
```
typedef int point[2];

int getX(point p) { return point[0]; }
void rmove(point p, int dx, int dy) {
    p[0] += dx; p[1] += dy; }
```

Object

- Simula, Smalltalk, CLOS, C++, Self, …
  - Dynamic dispatch
    - 適応可能な関数（メソッド）は
      オブジェクトごとに決まっている。型ではない。
  - Inheritance
    - コードの再利用（差分実装）
    - GUI ライブラリが killer app.

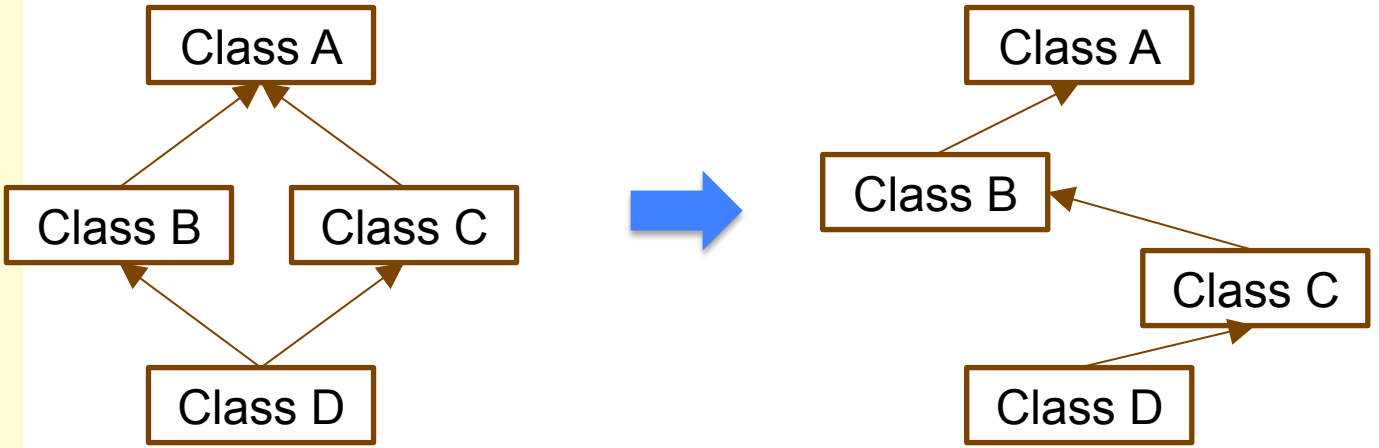
Multiple Inheritance

- もっと再利用！
  - 複数の部品の組み合わせで目的のクラスを定義
  - 素直で自然な拡張?
  - うまくゆかない例



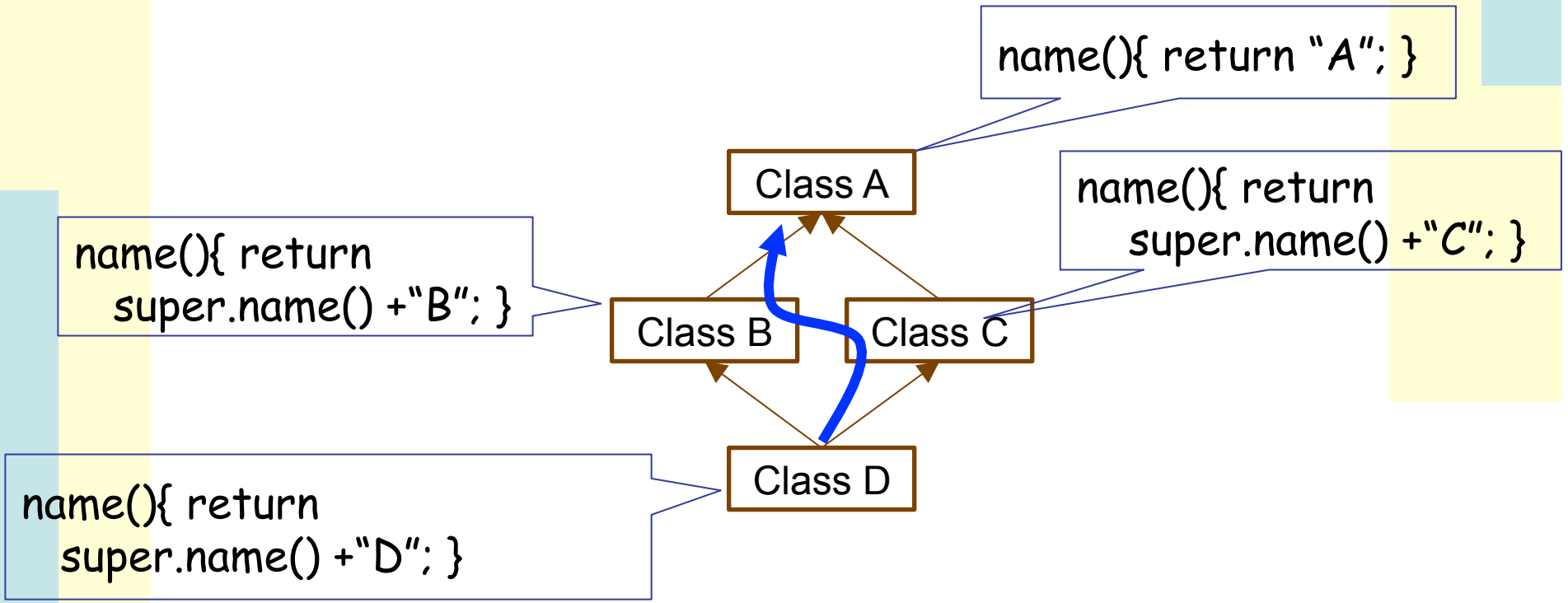
Mixin

- クラス
  - 複数の親クラスをもつ
    - Or 単一の親クラス＋複数 mixin
    - Mixin は instance を作れない
  - 複数の親クラスは並べ直されて単一継承
    - 並べ替え規則はさまざま



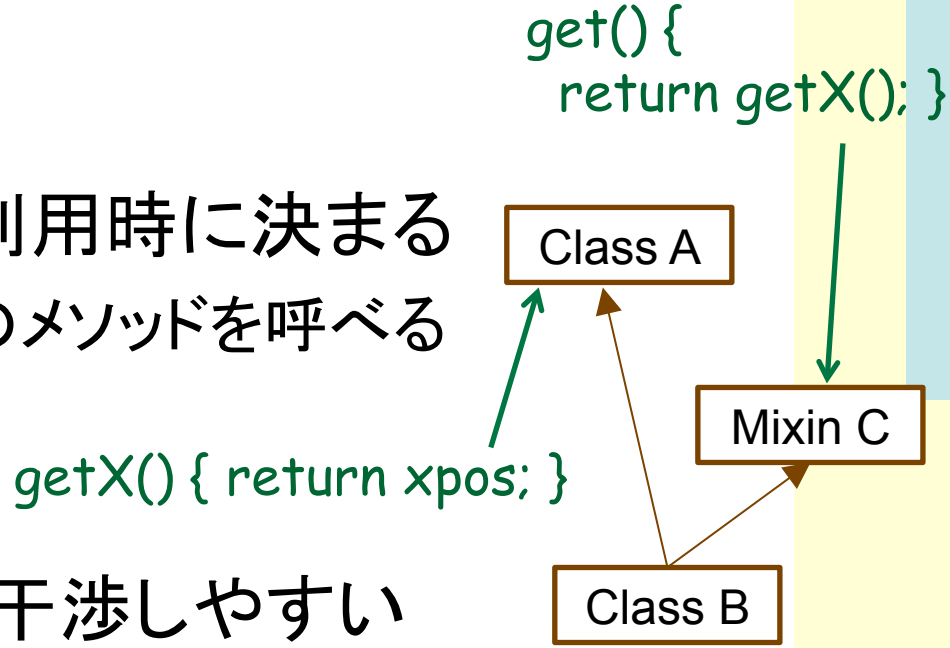
先ほどの例

- name()
  - super.name() の意味が明確



Mixin の特徴と欠点

- 継承
  - 親クラスは mixin 利用時に決まる
    - mixin からその親のメソッドを呼べる
  - 複数の mixin 間で干渉しやすい
    - 同名のメソッドを宣言したら？
    - 同名のメソッドを上書きしたら？
      - super 呼び出しの意味は？



Traits [2003]

- 再利用目的のメソッドの集まり
  - 状態変数 (fields) を持たない（後にもてるように変更）
  - Require メソッド
    - Java の abstract メソッドに相当?
  - Traits 間の干渉は、traits を継承するクラスが明示的に解消
    - Aliasing
    - Exclusion(除去）して上書き

Traits の例

super class

```
class Position {
    int x, y;
    int getX(); }
```

trait

```
class TCircle {
    require int getR();
    int size() {
        return getR() * getR() * PI;
    }
}
```

class

```
class Circle extends Position uses TCircle {
    int r;
    int getR() { return r; }
}
```

Flattening property

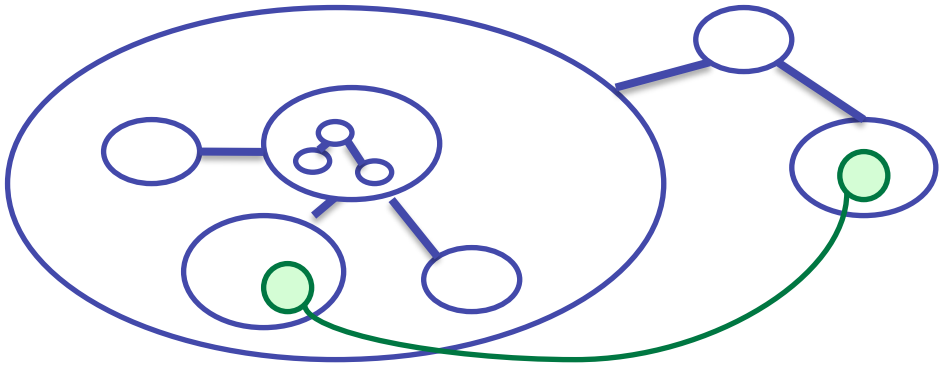
- 単一継承と上位互換
  - Class = Superclass + State + Traits + Glue
    - State, Traits, Glue を子クラスで直接まとめて宣言する場合と同じ
  - 優先順位
    - Class のメソッド、Traits のメソッド、最後に親クラスのメソッド
    - Traits は複数あっても同順位。
  - 細粒度の再利用が可能（細粒度すぎる？）

Glue code

- Multiple inheritance
- Mixin
  - Glue code は自動生成（見えない）。
  - Composition に柔軟性がない
- Traits
  - Glue code は手動記述
  - 柔軟な composition が可能

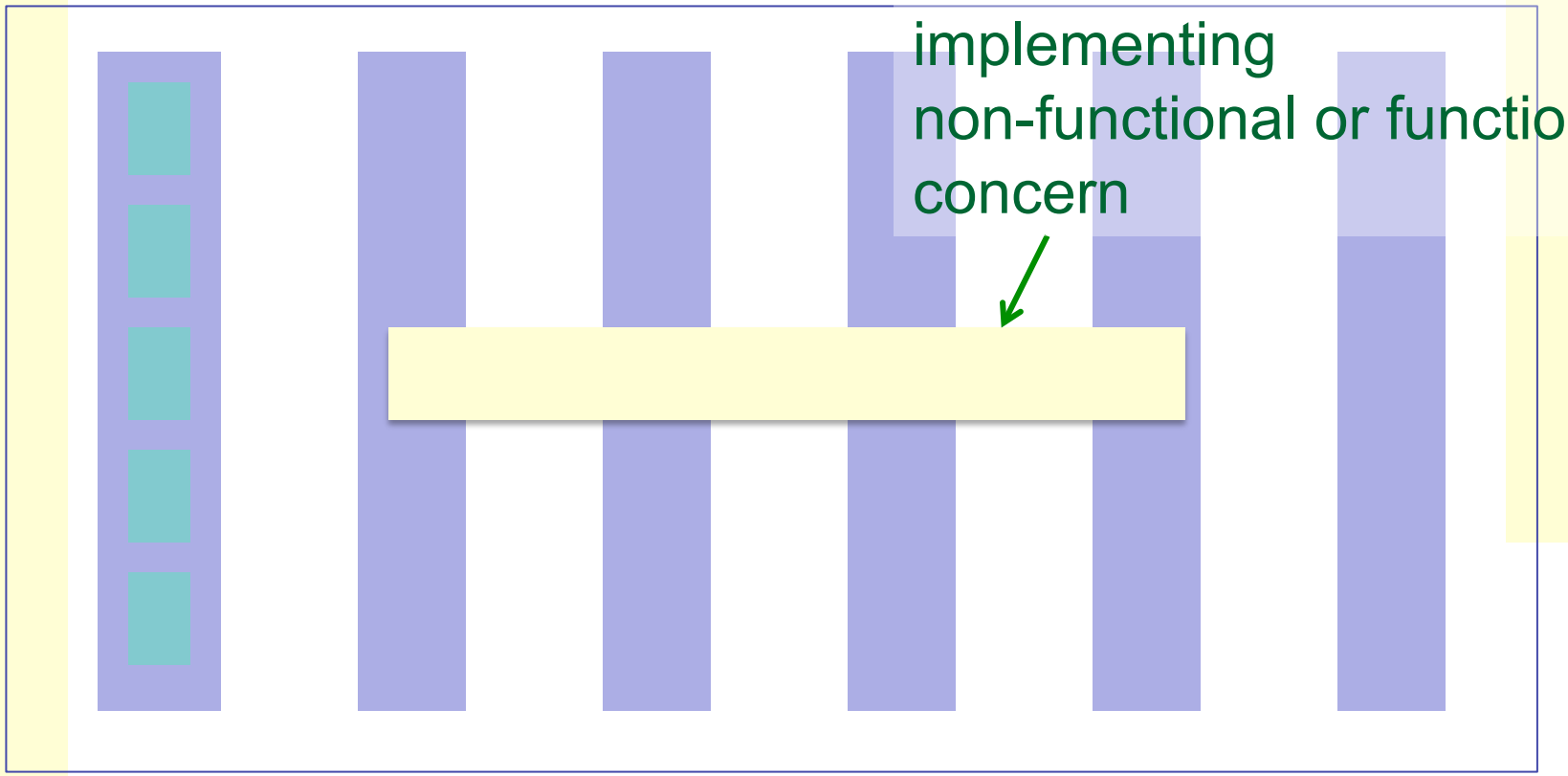
Aspects [late 90's]

- 新しいモジュール間結合
  - 階層的な module decomposition では不十分
  - Crosscutting concerns（横断的関心事）
    - 異なるモジュールに同時に属すべきコードが存在する
      - そのコードをひとつのモジュールに集めると、それまでまとまっていた別なコードが異なるモジュールに散らばってしまう。



Crosscutting concern

- Is a program hierarchically decomposed?



Logging (OOP/delegation)

- Logging code
  - scattering in many classes
  - Who instantiates Log?

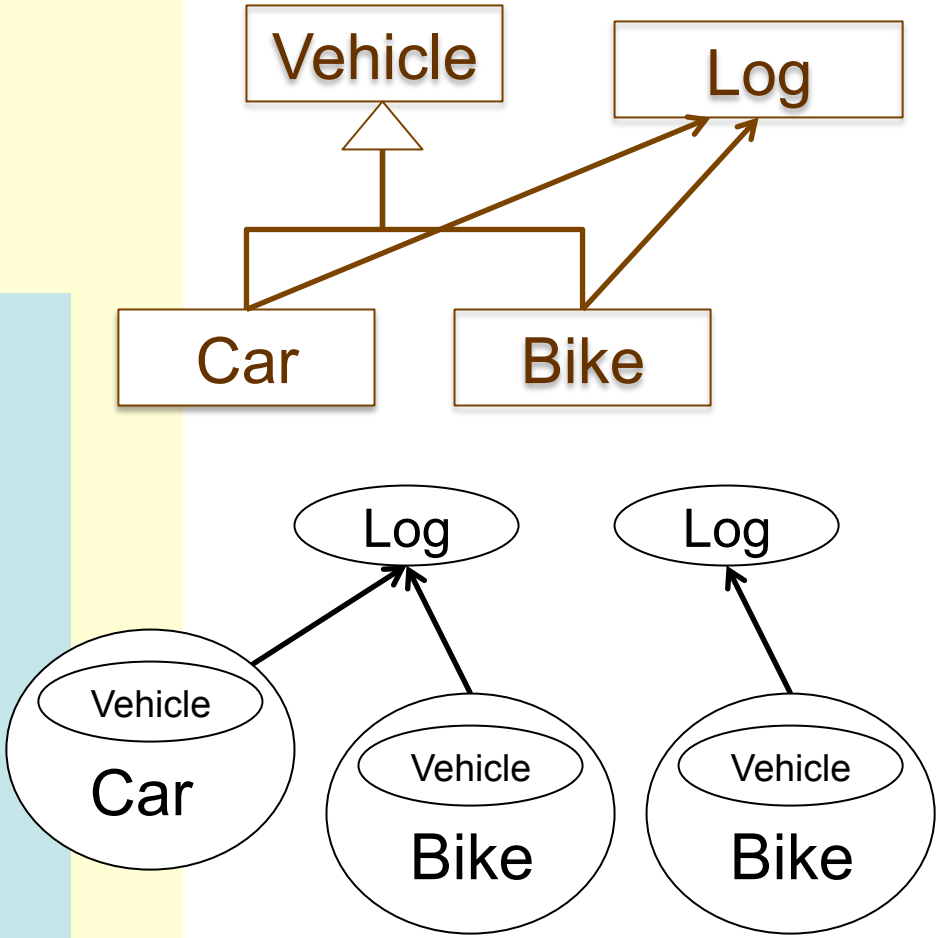
```
class Log {

    PrintStream output = System.out;
    void print(String msg) {
        output.println(msg);
    }
}
```

```
class Car extends Vehicle {
    Log log = …;
    void start() {
        log.print("start Car");
        …
    }
    void goback() {
        log.print("Car goes back");
        … }
}
```

```
class Bike extends Vehicle{
    Log log = …;
    void start() {
        log.print("start Bike");
        …
    }
}
```

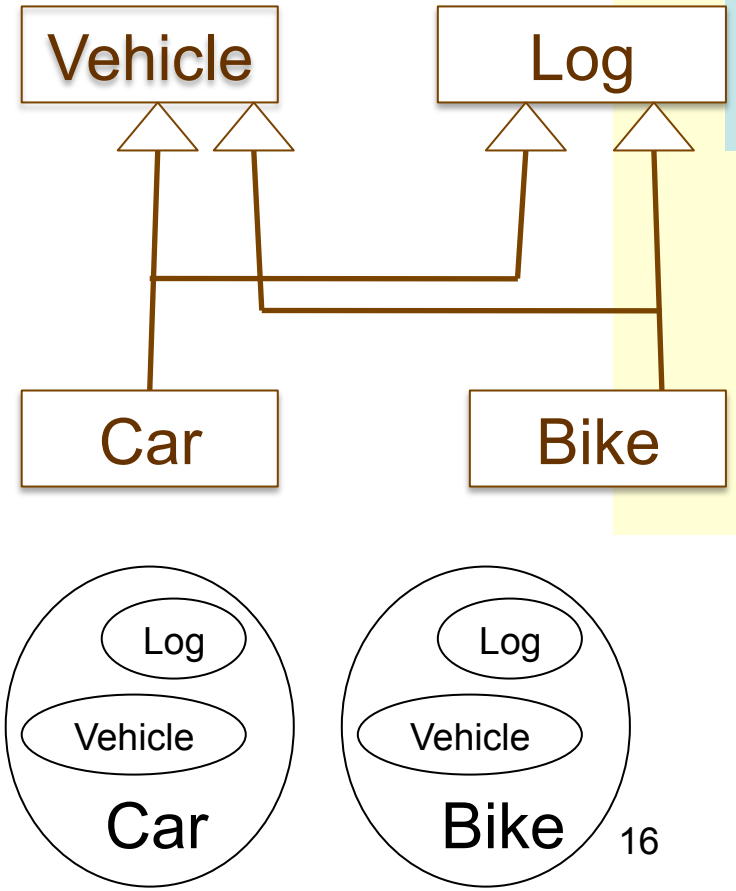
Instance-level composition

Real



Ideal?

(hierarchical composition)



2つの階層構造：has-a と is-a

- has-a 関係
  - 別オブジェクトとしてもつ
  - delegation を利用
- is-a 関係
  - スーパークラスとしてもつ
  - 継承を利用

```
class Pos {
    int x, y;
    int getX() { return this.x; }
}

class Pos3D {
    Pos p = new Pos();
    int z;
    int getX() { return p.getX(); }
}
```

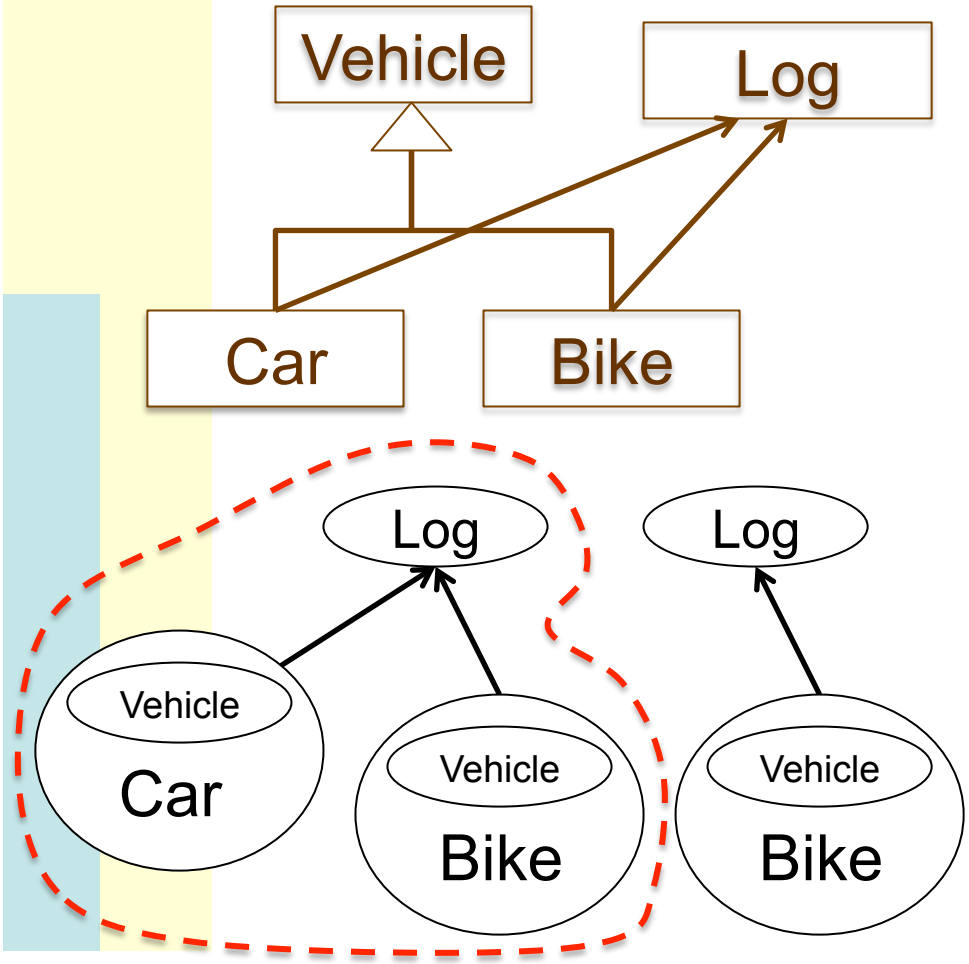
```
class Pos3D extends Pos {
    int z;
    int getX() { return super.getX(); }
}
```

Delegation

- より柔軟な has-a
- 仕事を他のオブジェクトに委任
  - JavaScript の prototype の動作
    - 自分にないメソッドなら prototype へ委任
  - Java でも可能
    - void move(int x, int y) {
      delegate.move(x, y);
      }
    - 委任先を動的に変えられれば、継承より強力
      - 使い方を誤ると、プログラムが複雑になる危険

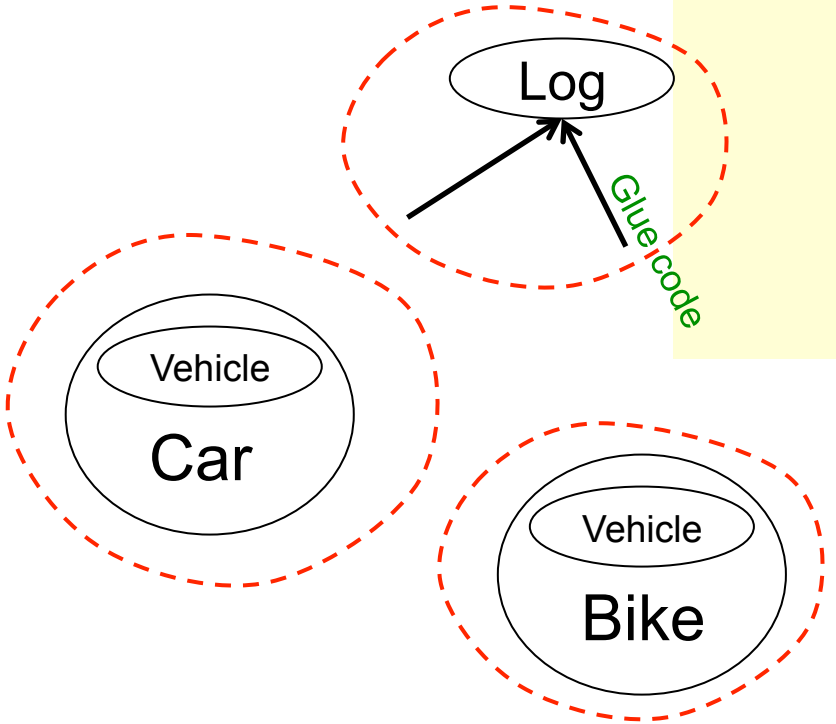
Modular glue code

All in one



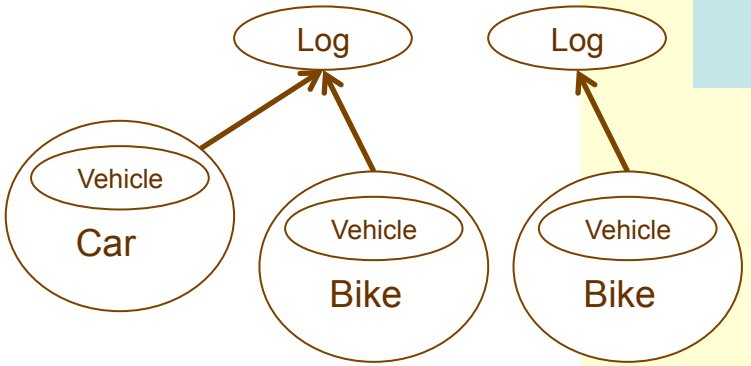
Not possible:

Tree separate components



Crosscutting State-Share

- Not useful
  - Is-a and Multiple inheritance
    - Mixin, traits
  - Has-a/part-of



- Delegation?
  - Composition (= writing glue code by hand) is annoying
  - need syntax support like Traits?

Scattering Glue code

- Is scattering even if no crosscutting share.
  - No modularity?
- Writing glue code is annoying

```
class Car extends Vehicle {
    Log log = …;
    void start() {
       log.print("start Car");
       …
   }
   void goback() {
       log.print("Car goes back");
       … }
}
```

```
class Log {

    PrintStream output = System.out;
    void print(String msg) {
        output.println(msg);
    }
}
```

```
class Bike extends Vehicle{
   Log log = …;
   void start() {
     log.print("start Bike");
      …
   }
}
```

Logging concern

- How a log message is printed
  - Console, a log file, syslog server, …

Log.print()

- Which messages are in the same log
  - Who share the Log object.
- When a log message is printed
  - start(), goback(), …
- What message is printed
  - “Start Car”, “Bike goes back”, …

Glue code

How to compose

- Glue code
  - 結合する部品(継承するクラス)の指定
  - インスタンスの生成
  - Delegation の指定
    - 必要なら名前衝突の解消
- どこへ書くか
  - 関連するオブジェクトのいずれか？
  - Shared state をもつオブジェクト？
  - Mixin, Traits など
    - それらを継承するサブクラスの中。

Logging (AOP)

- Logging code
  - in a single module
  - with glue code

```
aspect Log {

    "instantiate this once"

    "call this print() before every
    method invocation"

    PrintStream output = System.out;
    void print(String msg) {
        output.println(msg);
    }
}
```

```
class Car extends Vehicle {
    Log log = …;
    void start() {
      log.print("start Car");
        …
    }
    void goback() {
      log.print("Car goes back");
        … }
}
```

```
class Bike extends Vehicle{
    Log log = …;
    void start() {
      log.print("start Bike");
        …
    }
}
```

Definition of an aspect

- Logging aspect in AspectJ
  - Turning logging on/off does not need source code modification.

```
aspect Logging issingleton {
  PrintStream output = System.out;
  before(): execution(* Vehicle+.*(..)){
    output.println(thisPointcut.toString());
  }
}
```

Glue code in AOP (1)

暗黙のうちに
インスタンス生成

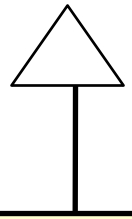
- Glue code をどこへ書くか
  - Shared states をもつオブジェクト
    - つまり aspect の中。Car や Bike クラスの中ではない。
- コードの再利用性
  - 強力な glue code 言語（pointcut 言語）
    - パターンマッチ
  - 非機能的関心事
    - logging, transaction, security, …
    - コードの再利用が可能に
      - 色々なメソッドの前後に機能追加が必要

Glue code in AOP (2)

- Glue コードの再利用性
  - 本質的に再利用が困難
  - Aspect を 2 つに分割 （継承の利用）
    - Abstract aspect の利用
    - 再利用性のない Glue code は sub-aspect に
      - Traits に似ているが、通常 states は super aspect におく。

Logging (AOP+inheritance)

```
aspect Log {
    PrintStream output = System.out;
    void print(String msg) {
        output.println(msg);
    }
}
```



```
aspect VehicleLog extends Log {

    "instantiate this once"

    "call this print() before every
    method invocation"

}
```

```
class Car extends Vehicle {
    Log log = …;
    void start() {
      log.print("start Car");
        …
    }
    void goback() {
      log.print("Car goes back");
        … }
}
```

```
class Bike extends Vehicle{
    Log log = …;
    void start() {
      log.print("start Bike");
        …
    }
}
```

非侵襲性 non-invasive (oblivious)

- Glue code をどこへ書くか
  - Shared states をもつオブジェクト
    - つまり aspect の中。Car や Bike クラスの中ではない。
- コードの保守性（独立性）の向上 (?)
  - アスペクトを書くだけでプログラムを拡張
    - Car や Bike クラスは変更なし
    - Car や Bike オブジェクトの生成も変更なし
      - Car を拡張して EcoCar を作成したら、
        new Car() を new EcoCar() に置換する必要がある。

新旧クラスの共存

- 継承、mixin、traits、…
  - 旧クラス：　親クラス
    新クラス：　子クラス（親クラスの機能拡張版）
  - オブジェクト生成時にクラス名で区別
    - 部品の機能拡張に便利
- AOP with non-invasiveness
  - 新クラスのみ有効
    - 横断的ゆえ、新旧を区別して共存させるのは困難
      - 既存ソフトウェアの拡張に便利
    - Classbox

Application of AOP

- JastAdd
  - コンパイラ･フレームワーク in Java
  - Aspect（非侵襲的に拡張）
    - Lexical analyer, parser, code generator の各拡張をまとめて 1 ファイルに
    - 属性文法を利用
    - Java 1.4 から差分記述だけで Java 1.5 対応
    - Java 1.5 から … AspectJ コンパイラ

Limitations of AOP

- Crosscutting concerns are real.
  - not only logging but functional extensions
- AspectJ is not the best.
  - Fragile pointcut problem
    - Too powerful glue code, no static check
  - Interferences between multiple aspects
    - No explicit control of composition
  - Breaking encapsulation?
    - Maintainability is low
    - Invasiveness

Modularity v.s. 非侵襲性

- 非侵襲性 (non-invasiveness)
  - 全ての関連コードを完全に1カ所に分離
    - ある意味、非常に modular?
      - JastAdd の例など
  - Modularity がない?
    - 新旧版が共存できない
      - 知らぬ間に新版が有効 — カプセル化の破壊
    - Separate compilation 不可
    - 複数 aspect の干渉
      - 同一メソッドを拡張すると誤動作しがち

Partial classes of C#

- Non-invasive extension
  - A class declaration can be divided into multiple partial declarations.

```
public partial class Position { int x; }
public partial class Position { int y; }
```

  - A compiler lexically merges them.
    - no modular compilation.
    - Categories of Objective-C are similar but allow modular compilation by including header files.

Open classes [oopsla'00]

- Non-invasive extension
  - allows adding a new method to an existing class.
  - Top-level method declarations

```
public boolean Node.typeCheck() { … }
```

    - similar to intertype declarations of AspectJ
    - Access control
      - only non-private methods are accessible.
    - Modularly type-checkable
      - no top-level method for interfaces or top-level abstract method

Feature-Oriented Programming

- Feature
  - ソフトウェアの個々の機能
    - 基本機能だけのソフトウェアに必要な feature を加えて目的のものにすることで開発
- AHEAD tool
  - Program = f(g(h(…k(c)…)))
    - f, g, h, …k: ある feature を追加するコード変換関数
    - c: 基本となるコード(を表す定数)
  - コード変換関数
    - 非侵襲 (non-invasive)
    - like mixin, open classes, intertype declaration,…
    - 上書きも可能

Context-Oriented Programming

- メソッドの上書き
  - 特定の実行文脈 (contexts) でのみ有効
    - ある種の dynamic scope

```
class Position {
    int x, y;
    void move(int dx, int dy) {
        … }
… }

foo(Position p) {
    p.move(3, 4);
}
```

```
aspect A {
    void Position.move(..) {
        … proceed(); …
    }
}

bar () {
  Position p = … ;
  foo(p);
  deploy(A) { foo(p); }
}
```

Context-Oriented Programming

- 特定の実行文脈 (contexts) でのみ有効な改変
  - 非侵襲: non-invasive
    - 上書きメソッドが有効になったら全オブジェクトで有効
      - 新旧版のオブジェクトの区別なし

- as in AOP
  - within, cflow pointcut
    - conditionally activate an advice
  - dynamic deploy
    - conditionally activate an aspect

モジュール機構の進化

- コード断片を結合する glue code の変遷
  - Glue code の設計と自動生成を模索してきた